Constructing an RDF Knowledge Graph
Knowledge Representation & Reasoning – Mod. 2
Artificial Intelligence 2021-2022 Final Project

[Giacomo Colosio]

1 Datasets and Task

The problem give us two different data-frames as cvs file, so two different tables which contain information about some musical album. Using pandas I will upload the cvs file as two Dataframes, so two-dimensional, size-mutable, potentially heterogeneous tabular data, or in more simpler word a table composed by rows and columns wich i will use to construct my RDF knowledge graph. So using pd.read_csv(csvfile.csv) i will generate df1 (dataframe 1) and df2(data frame 2).

The df1 has columns: *Year* which contains an integer value , *Album* which contains a string, *Artist* which contains a string , *Genre* which contains a string, *Subgenre* which contains a string.

df2 has as columns: *Ranking* which types is float, *Album* which contains a string, *Artist Name* contains a string, *Release Date* contains a string, *Genres* which contains a string, *Descriptors* contains a string Average Rating which types is float, Number of Ratings contains a string, Number of *Reviews* which types is integer.

Using this two data-frames I will generate my final data-frames called *df* with all the necessary information using as a join column *Albuminuscolo* wich . To join the dataframe i need to generate a column which we call [ALBUMminuscuolo] wich contain the album with only lowercase letter.

These two function calls are completely equivalent, in my project i have decide to use the second method and i have obtain the following final dataframe wich i called df:

I can notice that since the column Albuminuscolo of df1 and the same column for the df2 was the based for this merging this new dataframe df contains the rows of the two previous dataframes wich have the same album. Seeing the two initial dataframes we could observe that some columns probably could contains similar infotmation. This means that this merged dataframe contain some similar information in its comuln. So I decide to use only some columns of df, in particular: I use only the column [Artist] of df1 since it contains the same information of [Artist name] of df2. The same is for [Genre] and [Subgenere] from df1 and [Genres] from df2, so I ‘ve decided to only [Genre] and [Subgenres] from df1.

2 Workflow and RDF Knowledge Graph

The workflow follows 6 principal points:

1. Create an RDFLib Graph and add triples from the data of the dataframes.
2. Include a small ontology.
3. Generation of a taxonomy.
4. Load the RDF graph on a SPARQL Endpoint.
5. Materialize inferences in the endpoint's graph
6. Integrate my data with DBPedia's

2.1 Create an RDF Graph from the data of the data-frames

Using the RDFlib library I generate an RDF graph in wich i will add the information containing in df, the merge of the two inizial dataframes. The idea is to add the RDF-triple using as prefix the namespace ex, which permit the construction URI in an example domain.

This project contains different python libraries, in particular:

- rdf_lib which permit me to create a graph, to generate a triple, add a domain and range to properties, generate a taxonomies, add rdfs inference and other thing wich i show later.
- owlrl which simulates the OWL 2 (Web ontology lenguage) thanks to this i will generate all the inverse property, i will classificate the properties in object and data properties, i wil add some OWL inference and at the end i will connect this graph with DBPEDIA and it's components.

Using dataframe(df) i will generate the triples wich construct the initial rdf graph. This was based on the following triple:

As object Properties:

1. The relation *Album_Has_Artist* which connects each [Album] to its [Artist] and it's inverse.

```
#Add in the graph all the triples wich connect an [Album] to its [Genre], use ex.Album_Has_Genre property
for inded, row in df.iterrows():
   Album=row['Album']
   Album=Album.replace(' ','_')
   Album=Album.replace('"','')
   Album=Album.replace(' ','_')
   txt=row['Genre']
   genres = txt.split(',')
   for x in genres:
      if len(x)>1:
        u=x.replace(' ','')
        q=u.replace('&','/')
        z=q.replace(' ','')
        if '/' in z:
          z1=z.split('/')
          for y in z1:
            if len(y)>1:
              rdf_graph.add((URIRef(ex+str(Album)), URIRef(ex.Album_Has_Genre), URIRef(ex+y)))
        else:
          rdf_graph.add((URIRef(ex+str(Album)), URIRef(ex.Album_Has_Genre), URIRef(ex+z)))
rdf_graph.add((URIRef(ex.Album_Has_Genre), RDF.type ,OWL.ObjectProperty))
rdf_graph.add((URIRef(ex.Genre_Of_Album), RDF.type ,OWL.ObjectProperty))
rdf_graph.add((URIRef(ex.Genre_Of_Album), OWL.inverseOf ,(URIRef(ex.Album_Has_Genre))))
```

2. The relation *Album_Has_Genre* which connects each [Album] to each [Genre] and it's inverse.
3. The relation *Album_Has_subgenre* which connects each [Album] to each [Subgenre] and it's inverse.

Then for all of this propierties i will specifies with owlrl that they are object properties using the following triple: ex:[property] a owl:ObjectProperty .

As data Property:

1. The relation which connects each [Album] to its Name which is a string value.
2. The relation which connects each [Album] to its [Year] which is an integer value.
3. The relation which connects each [Album] to its [Release Date] which is a string value.
4. The relation which connects each [Album] to its [Ranking] which is an float value.
5. The relation which connects each [Album] to its [Descriptors] which is string value.

```
#Add in the graph all the triples wich connect an [Album] to its [Descriptors], use ex.AlbumDescriptor propert
for inded, row in df.iterrows():
   Album=row['Album']
   Album=Album.replace(' ','_')
   Album=Album.replace('"','')
   Album=Album.replace(' ','_')
   txt=row['Descriptors']
   subdescriptors= txt.split(',')
   for x in subdescriptors:
      if len(x)>1:
        z=x.replace(' ','')


        #Add the triple
        rdf_graph.add((URIRef(ex+str(Album)), URIRef(ex.AlbumDescriptor), URIRef(ex+z)))
   #print(rdf_graph.serialize())
rdf_graph.add((URIRef(ex.AlbumDescriptor), RDF.type , OWL.DatatypeProperty))
```

6. The relation which connects each [Album] to its [Average Rating] which is float value.
7. The relation which connects each [Album] to its [Number of ratings] which is string value.
8. The relation which connect each [Album] to its [Number of views] which is an integer value.

2.2 Include a small ontology

Now Using RDFlid i will add Hierarchical class structures and Domain and Range restrictions on properties. So, in a more detailed way I add:

As Hierarchical class structures:

1. The class Album and the i add all the value of [Album] column using inferences.
2. The class Artist and then i add all the value of [Artist] column using inferences.
3. The class Genre and then i add all the values of [Genre] column using inferences.
4. The class Subgenres and then i add all the values of [subgenres] column using inferences.

```
rdf_graph.add((URIRef(ex.Album), (RDF.type) , (RDFS.Class)  ))
rdf_graph.add((URIRef(ex.Artist), (RDF.type) , (RDFS.Class)  ))
rdf_graph.add((URIRef(ex.Genre), (RDF.type) , (RDFS.Class)  ))
rdf_graph.add((URIRef(ex.Subgenre), (RDF.type)  , (RDFS.Class)  ))
```

And as Domain and Range restrictions on properties:

1. For the relation Album_Has_Artist we have as Domain [Album] and as range [Artist] and the inverse for it's inverse.
2. For the relation Album_Has_Genre we have as Domain [Album] and as range [Genre] and the inverse for it's inverse.
3. For the relation Album_Has_SubGenre we have as Domain [Album] and as range [Subgenre] and the inverse for it's inverse.

For the dataproperties is specify the Domain which is always [Album] and for the range we have already insert the Literal with its specific datatypes.

2.3 Generation of the taxonomy

In this section I add a particular taxonomy to my KB. A taxonomy is a classification structure for objects of a given domain. Usually hierarchies with tree-like structures, and in this particular case we a a mereology which consist in a part-of relation.

From my graph I will extract a genres taxonomy from the dataset and will include it in my RDF KG. To do it I will generate a class for each genre, and i instantiate the fact that all these classes are a subclass of the class Genres (The class which contain all the possible genres). Then for all the possible classes of genres I add the subclasses of subgenres. So as taxonomy I ‘ve generated the following triples:

- [album] rdf:type [genre].
- [genre] rdf:type rdfs:Class.
- [genre] rdfs:subClassOf Genre.
- [subgenre] rdfs:subClassOf [genre].

2.4 Load the RDF graph on a SPARQL Endpoint

In this part of the project I load my RDF graph on my local SPARQL endpoint. Then i will run a few SPARQL queries on the graph. To do it i will use the Insert Data query which permit me to transfer my rdf_graph on my local endpoint. Then I will make queries on my graph

```
sparql = SPARQLWrapper("http://localhost:8890/sparql")
sparql.addDefaultGraph("ex:509653")
sparql.method = 'POST'
```

```
#using the rdf triple i add it on sparql
for s,p,o in rdf_graph:
    sparql.setQuery(f"INSERT DATA {{{s.n3()} {p.n3()} {o.n3()}.}}")
    sparql.query()
```

The SPARQL Query Language can be used to formulate queries over RDF data ranging from simple graph pattern matching to complex queries. SPARQL queries contain a set of triple patterns called a basic graph pattern. To make a SPARQL query I need:

Prologue, wich contains the prefix.

- Query form (Select/Ask/Construct/Describe)
- Query pattern
- Solution Modifiers (not mandatory)

In particular to make queries with SPARQLwrapper i need to:

- set the query using sparql.setquery.
- set the format using sparql.setReturnFormat.
- decode and print the results.

Query 3: Set a query wich give as a result a top 10 of the album between the 1990 and 2010 in base of album ranking.

```
sparql.setQuery("""prefix ex: <http://example.org/> select distinct * where {?x ex:AlbumRanking ?y ;
                   ex:AlbumYear ?z FILTER (?z > 1990 && ?z <2010) } Order by ?y limit 10""")
```

```
sparql.setReturnFormat("csv")
results = sparql.query()
results = results.convert()
print(results.decode())
```

```
"x","y","z"
"http://example.org/OK_Computer",1.0,1997
"http://example.org/Kid_A",4.0,2000
"http://example.org/Loveless",6.0,1991
"http://example.org/In_Rainbows",15.0,2007
"http://example.org/Illmatic",21.0,1994
"http://example.org/Dummy",42.0,1994
"http://example.org/The_Low_End_Theory",55.0,1991
"http://example.org/The_College_Dropout",68.0,2003
"http://example.org/Aquemini",84.0,1998
"http://example.org/Funeral",93.0,2004
```

2.5 Materialize inferences in the endpoint's graph

In this section I use SPARQL insert queries ot materialize inferences that can be made from my KG. The inference which i can do are based in RDFS and OWL so i divide it into differents part, the first one foR RDFS inferences following the 13 rules, in the second one i use OWL inferences.

RDFS inferences.

Using the RDFS entailment rules i will say that:

First thanks to the rules RDFS-2 and RDFAS-3 it's possible to say:

- All the entities of the [Album] column are of type:Album.
- All the entities of the [Artist] column are of type:Artist.
- All the entities of the [Genre] column are of type:Genre.
- All the entities of the [SubGenre] column are of type:Subgenre.

This because for the property: Album_Has:Artist we need to have: as a domain an entity of type (so class) Album. As a range an entity of type (so class) Artist. The same idea works for Album_Has_Genre and Album_Has_Subgenre.

Then at the same way I use the entailment rule: RDFS4, RDFS5, RDFS7, RDFS9, RDFS11.

Now thanks to the rules RDFS7 it's possible to say:

- if A is a subPropertyOf B. and X A Y then X B Y .

```
sparql.setQuery("prefix ex: <http://example.org/>  insert {?x ?b ?y}  where {?a rdfs:subPropertyOf ?b.  ?x ?a ?y }")
sparql.setReturnFormat("csv")
results = sparql.query()
results = results.convert()
```

OWL inferences.

OWL , or ontology web language , is a Semantic Web language for expressing ontologies designed so that its semantics can be defined via a translation into a Description Logic (DL). An ontology is a set of precise descriptive statements about some part of the world (usually referred to as the domain of interest). It's composed by a terminological and an assertional box. Using OWL i can add some inferences to my knowledge, i have decide to add:

- All the triple using the owl:inverse function which i have generated in the part 2 of the project

Now add: the triple using the owl:inverse function which i have generated in the part 2 of the project

```
#first owl inference, add all the inverse relation of the object properties
sparql.setQuery("prefix ex: <http://example.org/>  insert {?x ?i ?y} where {?y ?z  ?x. ?i owl:inverseOf ?z}")
sparql.setReturnFormat("csv")
results = sparql.query()
results = results.convert()
```

2.6 Integrate my data with DBPedia's

Using the ex:AlbumName of album names and bands in the dataframes to create a relation with DBPedia entities, then and add triples about these DBPedia IRIs to your graph. In this part I:

- Connect all the ex:Artist with the equivalent entity in DBpedia.
- Connect all the ex:Album with the equivalent entity in DBpedia.
- Connect all the ex:genre with the equivalent entity in DBpedia.

I also add information from DBPedia to your graph with a federated query, so using SPARQL insert {} where i add: For more album as possible:

- Add dbr:producer for each album.

The problem wich i have to limitate with this integration technique is that for some album name the dpedia link which I create is not talking about of what I want(e.g the album) but its information correspond to something else. So to optimize the process I count the word Album or Artist in the page to be sure that the dbr.[AlbumName or Artist] is connected with my intended interpretation.

To do it I use the following codes:

```
#connect all the artist entity with owl:sameAs with the entities in dbpedia
from urllib.request import urlopen
from urllib.error import *
dbr = Namespace('http://dbpedia.org/resource/')
rdf_graph.bind('dbr', dbr)
#connect all the album entity with owl:sameAs with the entities in dbpedia
print('iniziato caricamento')
for inded, row in df.iterrows():
    Artist=row['Artist']
    Artist = Artist.replace(u'\xa0', u' ')
    Artist=Artist.replace(' ','_')


    try:
        html = urlopen(URIRef(dbr+str(Artist)))


    except HTTPError as e:
        print("HTTP error", e)

    except URLError as e:
        print("Opps ! Page not found!", e)

    else:
        x=(requests.get(URIRef(dbr+str(Artist))).text.count('Artist'))
```

```
#connect all the album entity with owl:sameAs with the entities in dbpedia
print('iniziato caricamento')
import re
import requests

dbr = Namespace('http://dbpedia.org/resource/')
rdf_graph.bind('dbr', dbr)
for inded, row in df.iterrows():
    Album=row['Album']
    Album=Album.replace(u'\xa0', u' ')
    Album=Album.replace(u'\xe2', u' ')
    AlbumTitle=Album.replace(' ','_')
    AlbumTitle=AlbumTitle.replace("","")
    try:
        html = urlopen(URIRef(dbr+str(AlbumTitle)))
        page=urlopen(URIRef(dbr+str(AlbumTitle))).read()

    except HTTPError as e:
        print("HTTP error", e)
    except URLError as e:
        print("Opps ! Page not found!", e)
    else:
        x=(requests.get(URIRef(dbr+str(AlbumTitle))).text.count('Album'))
        if x>=10:
            print(URIRef(ex+AlbumTitle), OWL.sameAs, URIRef(dbr+AlbumTitle))
            print(x)
            rdf_graph.add(((URIRef(dbr+AlbumTitle)), (OWL.sameAs), (URIRef(ex+AlbumTitle))))
print('finito caricamento')
```

As a last thing I add all the producer of the ex:Album and the rdfs;label to ex:Artist. To do it I will use a federated query. Federated queries are queries that are distributed over different SPARQL endpoints. With it we can retrieve RDF data from multiple SPARQL endpoints such as DBPedia and Wikidata and integrate it with the data on your SPARQL endpoint.

My federated queries have the following structures:

```
In [224]: sparql.setQuery("""


    prefix ex: <http://example.org/>

    prefix dbp: <http://dbpedia.org/property/>

insert{ graph <ex:509653> {?x rdfs:label ?name }}
where {
        graph <ex:509653> { ?x owl:sameAs ?artist; a ex:Artist}

        service <https://dbpedia.org/sparql> { ?artist rdfs:label ?name}
    }

""")
sparql.setReturnFormat("csv")
results = sparql.query()
results = results.convert()
print(results.decode())
#print(results.convert().toxml())
```

3 Conclusion

In this project i have generated an rdf graph wich contains the explicit and implicit information contained in two dataframes. It could be expanded in a lot of ways, some idea could be:

- Add subclasses to differentiate Artist in Groups and single artist taking information from dbpedia.
- Add some information from another source that is not dbpedia, such as wikidata.
- Add subclasses as equivalent class of already existing class combined with Boolean constructors.
- Specify wich artist are alive and which artist are dead with owl:NegativePropertyAssertion (To specify for In particular wich Paul MCcartney is still Alive against the conspirational theories).

In general all the program contains some assumption but the big one is in the section 2.5 in which I integrate my graph using dbpedia, to do it I use a method which presents two major weaknesses:

- firstly it's requires a lot of time because the code search and verify the exitence of all possible URL
- Secondly, I can't be sure of the fact that if in the URL the world Album is present, the URL is really the resource of that album on dbpedia (possible homonymy).